CS10720 Problems and Solutions

Thomas Jansen

Today: Sorting: Insertion Sort

February 19th

Plans for Today

Our Problem for Today

Remember searching in a sorted array (of size n)
can be in with binary search in time $\Theta(\log n)$ per search

What if the array is not sorted?

Observation can use linear search (without stopping early)
in time $\Theta(n)$ per search

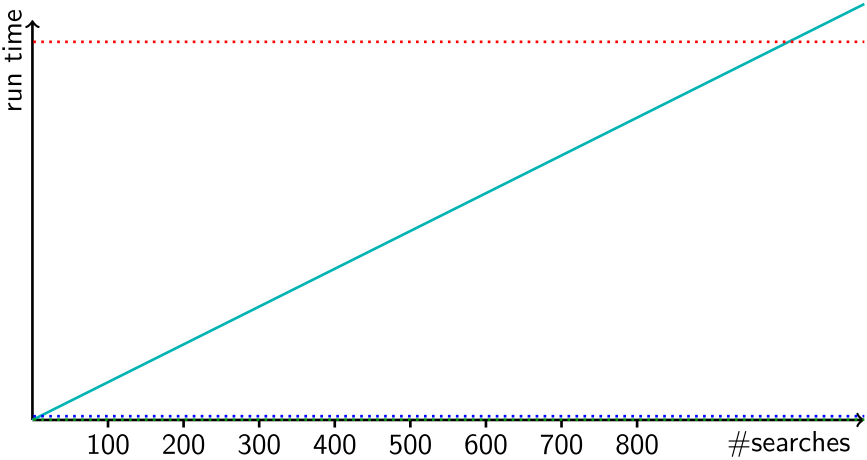
Problem have unsorted array of size n
have s searches to perform
(potentially both, n and s, very large)

Potential solutions

- Perform s linear searches.
  worst case time $\Theta(s \cdot n)$
- 1 Sort the array. 2 Perform s binary searches.
  worst case time $\Theta(\text{Time for sorting} + s \cdot \log n)$

Time for s Searches with Linear and Binary Search

Example array size 1000, time for linear s. (solid) and binary s. (dotted) with time for sorting n, $n\log_2(n)$, n^2



Sorting an Array

Have array of size n with keys, unsorted

Want the same array with the same keys but sorted in ascending order

Remark 'the same array' is a stricter requirement than 'an array of equal size'
implies no extra space, sorting within the same array

No extra space at all? Well, a little is okay...

Definition

A sorting algorithm that sorts an array of size n while using extra space at most $O(\log n)$ is called in situ.

Towards an In Situ Sorting Algorithm

Have array of size n with keys, unsorted

Want the same array with the same keys but sorted in ascending order

Let's start small!

What if the size of the array is 0?

Observation size 0 means 'the empty array'
empty array is sorted (at least it's not unsorted)

What if the size of the array is 1?

Observation size 1 means 'exactly one element'
array with one element is sorted

Idea for an In Situ Sorting Algorithm

Remember arrays of size ≤ 1 are already sorted

Idea sorted array of size k can be increased to size $k+1$
by finding position of `keys[k]` in `keys[0..k-1]`
and placing it there, potentially shifting items to the right

```
void sort(long *keys, long size) {
  long k; /* indicates end of sorted (partial) array) */
  long pos; /* position of new item */
  long key; /* new item */
  long i; /* used to move items */
  for ( k=0; k<size-1; k++ ) { /* increase sorted array to position k+1 */
    pos = search(keys, keys[k+1], k+1); /* find position for keys[k+1] */
    key = keys[k+1]; /* remember item before overwriting it */
    for ( i=k; i>=pos; i-- ) { /* move items 1 to the right */
      keys[i+1] = keys[i];
    }
    keys[pos] = key; /* place new item */
  }
}
```

Remember versions in English and pseudo-code in lecture notes

Towards Insertion Sort

```
void sort(long *keys, long size) {
  long pos; /* position of new item */
  long key; /* new item */
  for ( long k=0; k<size-1; k++ ) { /* increase sorted array to k+1 */
    pos = search(keys, keys[k+1], k+1); /* find position for keys[k+1] */
    key = keys[k+1]; /* remember item before overwriting it */
    for ( i=k; i>=pos; i-- ) { /* move items 1 to the right */
      keys[i+1] = keys[i]; /* move items 1 to the right */
    keys[pos] = key; /* place new item */
  }
}
```

Observation needs `long search(long *keys, long key, long size)`

- `keys` is array of keys, starting at 0, ending at `size-1`
- `key` is new key
- returns position `pos` of `key` in `keys`
  such that `keys[pos-1]` $\leq$ `key` $\leq$ `keys[pos]`
  or `pos=0` if `key` $\leq$ `keys[0]`
  or `pos=size` if `keys[size-1]` $\leq$ `key`

Remark not a precise definition
allows some flexibility with equal keys

Implementing `long search(long *keys, long key, long size)`

Remember need `long search(long *keys, long key, long size)`

- `keys` is array of keys, starting at 0, ending at `size-1`
- `key` is new key
- returns position `pos` of `key` in `keys`
  such that `keys[pos-1]` $\leq$ `key` $\leq$ `keys[pos]` or
  `pos=0` if `key` $\leq$ `keys[0]`, or `pos=size` if `keys[size]` $\leq$ `key`

Remember linear search in a sorted array

```
int search(int *keys, int size, int key) {
  int i; /* index variable for a loop over keys */
  i=0; /* start searching at the beginning */
  while ( (i<size) && (keys[i]<key) ) { /* search for key */
    i++; /* move to next item */
  }
  if ( (i==size) || (keys[i]>key) ) /* 2 cases for key not found */
    return -1;
  else
    return i;
}
```

Observation almost what we need

Adapting `long search(long *keys, long key, long size)`

Remember need `long search(long *keys, long key, long size)`

- `keys` is array of keys, starting at 0, ending at `size-1`
- `key` is new key
- returns position `pos` of `key` in `keys`
  such that `keys[pos-1]` $\leq$ `key` $\leq$ `keys[pos]` or
  `pos=0` if `key` $\leq$ `keys[0]`, or `pos=size` if `keys[size]` $\leq$ `key`

```
int search(int *keys, int size, int key) {
  int i; /* index variable for a loop over keys */
  i=0; /* start searching at the beginning */
  while ( (i<size) && (keys[i]<key) ) { /* search for key */
    i++; /* move to next item */
  }
  if ( i==size ) /* all items < key */
    return size;
  return i; /* i is correct position */
}
```

Remember we can search faster in a sorted array with binary search

Implementing search With Binary Search

Remember need `long search(long *keys, long key, long size)`

- keys is array of keys, starting at 0, ending at size-1
- key is new key
- returns position pos of key in keys
  such that keys[pos-1] $\leq$ key $\leq$ keys[pos] or
  pos=0 if key $\leq$ keys[0], or pos=size if keys[size] $\leq$ key

Remember binary search in a sorted array

```
int search(int *keys, int size, int key) {
  int left, right; /* defining boundaries of search area */
  int middle; /* the 'middle' where we hope to find the key */
  left = 0; /* leftmost key */
  right = size-1; /* rightmost key */
  while ( left <= right ) {
    middle = left + (right-left)/2; /* midpoint */
    if ( keys[middle] == key )
      return middle;
    if ( keys[middle] < key ) /* need to look in right half */
      left = middle+1; /* can exclude middle */
    else /* need to look in left half */
      right = middle-1; /* can exclude middle */
  }
  return -1;
}
```

Observation almost what we need

Adapting `search` With Binary Search

Remember need `long search(long *keys, long key, long size)`

- `keys` is array of keys, starting at 0, ending at `size-1`
- `key` is new key
- returns position `pos` of `key` in `keys`
  such that `keys[pos-1]` $\leq$ `key` $\leq$ `keys[pos]` or
  `pos=0` if `key` $\leq$ `keys[0]`, or `pos=size` if `keys[size]` $\leq$ `key`

Remember binary search in a sorted array

```
int search(int *keys, int size, int key) {
  int left, right; /* defining boundaries of search area */
  int middle; /* the 'middle' where we hope to find the key */
  left = 0; /* leftmost key */
  right = size-1; /* rightmost key */
  while ( left <= right ) {
    middle = left + (right-left)/2; /* midpoint */
    if ( keys[middle] == key )
      return middle;
    if ( keys[middle] < key ) /* need to look in right half */
      left = middle+1; /* can exclude middle */
    else /* need to look in left half */
      right = middle-1; /* can exclude middle */
  }
  return left;
}
```

Insertion Sort

```
void sort(long *keys, long size) {
  long pos; /* position of new item */
  long key; /* new item */
  for ( long k=0; k<size-1; k++ ) { /* increase sorted array to k+1 */
    pos = search(keys, keys[k+1], k+1); /* find position for keys[k+1] */
    key = keys[k+1]; /* remember item before overwriting it */
    for ( long i=k; i>=pos; i-- )
      keys[i+1] = keys[i]; /* move items 1 to the right */
    keys[pos] = key; /* place new item */
  }
}
int search(int *keys, int size, int key) {
  int left, right; /* defining boundaries of search area */
  int middle; /* the 'middle' where we hope to find the key */
  left = 0; /* leftmost key */
  right = size-1; /* rightmost key */
  while ( left <= right ) {
    middle = left + (right-left)/2; /* midpoint */
    if ( keys[middle] == key )
      return middle;
    if ( keys[middle] < key ) /* need to look in right half */
      left = middle+1; /* can exclude middle */
    else /* need to look in left half */
      right = middle-1; /* can exclude middle */
  }
  return left;
}
```

Analysing Insertion Sort (#Key Comparisons)

```
void sort(long *keys, long size) {
  long pos; /* position of new item */
  long key; /* new item */
  for ( long k=0; k<size-1; k++ ) { /* increase sorted array to k+1 */
    pos = search(keys, keys[k+1], k+1); /* find position for keys[k+1] */
    key = keys[k+1]; /* remember item before overwriting it */
    for ( long i=k; i>=pos; i-- )
      keys[i+1] = keys[i]; /* move items 1 to the right */
    keys[pos] = key; /* place new item */
  }
}
```

For simplicity only worst case analysis
and only number of comparisons of keys

Observations

- main loop goes over $n-1$ values ($n =$ size)
- binary search worst case time $O(\log(k+1)) = O(\log n)$

Worst Case **Number of Comparisons**

$$O(\log 1) + O(\log 2) + O(\log 3) + \cdots + O(\log n)$$

$$= \sum_{i=1}^{n} O(\log i) = n \cdot O(\log n) = O(n \log n)$$

Analysing Insertion Sort

```
void sort(long *keys, long size) {
  long pos; /* position of new item */
  long key; /* new item */
  for ( long k=0; k<size-1; k++ ) { /* increase sorted array to k+1 */
    pos = search(keys, keys[k+1], k+1); /* find position for keys[k+1] */
    key = keys[k+1]; /* remember item before overwriting it */
    for ( long i=k; i>=pos; i-- )
      keys[i+1] = keys[i]; /* move items 1 to the right */
    keys[pos] = key; /* place new item */
  }
}
```

For simplicity only worst case analysis

Observations

- main loop goes over $n-1$ values ($n =$ size)
- binary search worst case time $O(\log(k+1)) = O(\log n)$
- inner loop in the worst case $\Theta(k+1)$ values
- $\Theta(k)$ of inner loop dominates run time

Worst Case Run Time $\Theta(1)+\Theta(2)+\Theta(3)+\cdots+\Theta(n-1)$

$$= \sum_{i=1}^{n-1} \Theta(i) = \Theta(n^2)$$

Analysing Insertion Sort (continued)

```
void sort(long *keys, long size) {
  long pos; /* position of new item */
  long key; /* new item */
  for ( long k=0; k<size-1; k++ ) { /* increase sorted array to k+1 */
    pos = search(keys, keys[k+1], k+1); /* find position for keys[k+1] */
    key = keys[k+1]; /* remember item before overwriting it */
    for ( long i=k; i>=pos; i-- )
      keys[i+1] = keys[i]; /* move items 1 to the right */
    keys[pos] = key; /* place new item */
  }
}
```

For simplicity only worst case analysis

Observations

- main loop goes over $n-1$ values ($n =$ size)
- $\Theta(k)$ of inner loop dominates run time

Worst Case Run Time $\sum_{i=1}^{n-1} \Theta(i) = \Theta(n^2)$

Remark $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \Theta(n^2)$

but even simpler $\sum_{i=1}^{n-1} i < (n-1) \cdot (n-1) = \Theta(n^2)$

Analysis of Insertion Sort: Critical Appraisal

```
void sort(long *keys, long size) {
  long pos; /* position of new item */
  long key; /* new item */
  for ( long k=0; k<size-1; k++ ) { /* increase sorted array to k+1 */
    pos = search(keys, keys[k+1], k+1); /* find position for keys[k+1] */
    key = keys[k+1]; /* remember item before overwriting it */
    for ( long i=k; i>=pos; i-- )
      keys[i+1] = keys[i]; /* move items 1 to the right */
    keys[pos] = key; /* place new item */
  }
}
```

For simplicity only worst case analysis

Observations

- main loop goes over $n-1$ values ($n =$ size)
- $\Theta(k)$ of inner loop dominates run time

Worst Case Run Time $\sum_{i=1}^{n-1} \Theta(i) = \Theta(n^2)$

Observation linear search also worst case time $\Theta(k)$
insertion sort with linear and binary search equally fast in the worst case

Find out how that looks in the practicals!

Coming Back to the Original Motivation

Problem have unsorted array of size n and s searches to perform

Potential solutions

- Perform s linear searches.
  worst case time $\Theta(s \cdot n)$
- ① Sort the array. ② Perform s binary searches.
  worst case time $\Theta(\text{Time for sorting} + s \cdot \log n)$
  $= \Theta(n^2 + s \cdot \log n)$

Observation approach using search better if $s = \omega(n^2/(n - \log n))$
(and not worse if $s = \Omega(n^2/(n - \log n))$)

size n	'required searches' $n^2/(n - \log_2 n)$
10	≈ 15
100	≈ 107
1000	≈ 1010
10 000	$\approx 10\,013$
100 000	$\approx 100\,017$

Summary & Take Home Message

Things to remember

- motivation: searching in an unsorted array v sorting+searching in a sorted array
- in situ sort algorithms
- InsertionSort with linear and binary search
- analysis InsertionSort:
  - $\Theta(n^2)$ operations in the worst case
  - $\Theta(n \log n)$ comparisons in the worst case
- InsertionSort+search in a sorted array v searching in an unsorted array

Take Home Message

- Sorting is not much harder then searching.
- A much more efficient search may not help in the worst case if other operations dominate the run time.

Lecture feedback `http://onlineted.com`